Southern Software

ACCEL2

Compiler for TRS-80* BASIC

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

ACCEL2 COMPILER for TRS-80 BASIC: OPERATING INSTRUCTIONS.

ACCEL2 is an advanced version of ACCEL, Southern Software's compiler for TRS-80 BASIC. ACCEL will compile programs in the Leve12 language, while ACCEL2 will compile tire DISK BASIC extensions as well. (These extensions are sometimes referred to as Level3).

ACCEL2 occupies 5632 bytes at compile-time. This relatively low size is achieved by a technique of selective compilation. For instance I/O statements such as PRINTor INPUT are not translated at all but remain in the compiled program in their source form, and are executed by the resident BASIC interpreter. Statements involving INTEGERS and flow-of-control statements (GOTO, GOSUB, RETURN) are, by contrast, translated to directly-executed Z80 machine-instructions. Other, more complex statements are translated into calls to ROM routines. ACCEL2 selects more statements for translation than ACCEL, notably those involving STRINGS and SINGLE and DOUBLE data-types. ACCEL2 also translates references to one-dimensional array elements, and translates more functions than ACCEL.

ACCEL2 SUPPLIED ON TAPE OR WAFER.

If you have purchased ACCEL2 on diskette, skip the next four sections. Installation from EXATRON wafer is similar to installation from tape, and is covered in these sections.

The tape or wafer supplied is self-relocating. This gives you the freedom to load the compiler anywhere in memory you please. It also provides the freedom to make mistakes, so please check all address arithmetic carefully. You need only perform the installation operation once, and then you can take your own back-up copies on tape or disk, for subsequent direct loading.

You must load the tape supplied under Level2 (not DISK BASIC) using the SYSTEM command. It will load itself at locations 18944 and up, and then, under your control, will relocate itself to any chosen location above this. The compiler will not run correctly unless it is loaded in PROTECTED memory. Depending on how much RAM you have, and on what other machine-language programs you want resident, decide where you want to locate the compiler. The table overleaf gives addresses that are suitable if you want to load the compiler as high in memory as possible. For the main text, we will assume as an example that ACCEL2 is loaded on a 32K machine. In this case your answer to the initial MEMORY SIZE: question will be calculated as follows:

```
49152   (Upper limit of memory on a 32K machine)
-5632   (Size of ACCEL2 compiler)
-----
43520   = "M", answer to MEMORY SIZE?
-----
```

Notes:

1) On Video-Genie (PMC-80) you don't get the MEMORY SIZE? prompt. However, on power up the machine gives you the opportunity to enter a number after the first READY?. This is exactly the same number referred to as "M" in these examples.

2) If you are going to use TSAVE to make either a tape backup copy of ACCEL2, or to save the compiled program, then allow a further 512 bytes of protected memory, i.e. set M=43008 instead.

TABLE OF USEFUL MEMORY ADDRESSES.

This first table gives values of addresses you can use in order to locate ACCEL2 as high in your machine as possible, (assuming you have no other machine-code programs above it).

16K	32K	48K	
32767	49151	65535	Top-of-memory address
27136	43520	59904	Start of ACCEL2, MEM SIZE
26624	43008	59392	MEM SIZE,(room for TSAVE)
27136-32767	43520-49161	59904-65535	BACKUP range to save ACCEL2
27136-28415	43520-44799	59904-61183	Run-time routines, for tape
27136-28671	43520-45055	59904-61439	Run-time routines, if disk used

This second table gives address ranges you can use when compiling programs to sell on a smaller machine, They ensure that only the minimum amount of ACCEL2 (i.e. the run-time routines) resides in the end-user's smaller memory.

16K	32K	Target machine size
31488-37119	47872-53503	ACCEL2 address range, on your machine, TAPE only
31488-32767	47872-49151	Run-time routines, in end-user's machine
31232-21063	47616-53247	ACCEL2 address range, on your machine, if DISK used
31231-32767	47616-49151	Run-time routines, in end-user's machine

LOADING THE TAPE SUPPLIED.

To load the tape or wafer supplied, type responses as follows:

1) MEMORY SIZE? 43520 (enter) (or your value of "M").
   RADIO SHACK LEVEL II BASIC
   READY

2) SYSTEM (enter) or 2) @LOAD1 for EXATRON wafer.
   Now continue at step 6, below.

3) *? ACCEL2 (enter)

4) Tape loads

5) *? / (enter)

6) TARGET ADDR? (enter) (or any chosen protected address).
   READY

Notes:

1) The tape loading process at step 4 is a standard core-image tape load, and is subject to the usual variability on volume, head alignment, etc, A pair of asterisks will blink on the display, If no asterisks appear, or if a C is displayed, then there has been a loading error. Retry from step 1 with a different volume setting. Two copies are supplied on the tape, in case one gets damaged. Damage is almost invariably due to either a tape kink (a tiny fold in the tape) or to recorded noise, caused by RESETTING the computer in the middle of a tape load. (Always stop the cassette player before hitting RESET on a bad load),

2) Step 6 lets you relocate the compiler anywhere in protected memory. If you just hit enter, then it relocates to M, the answer to the MEMORY SIZE question. If you have other machine-code programs you want to hold concurrently in memory, then you may wish to respond with a value different from M.

3) If you have a Stringy Floppy controller active, then you may find that the MEMORY SIZE value has been modified by the activation process, In thus case type in the explicit value of TARGET ADDR, and do not rely on the default.

4) On Video-Genie, for (enter) read NEW LINE key.

SAVING A BACK-UP.

You can now save your own, fixed-location back-up copy of ACCEL2. This can be on tape, disk, or wafer. It will be shorter than the original file, but more important, it will load under either Level 2 or DISK BASIC, and it will load directly at its final location, without corrupting location 18944 and up.

A) Backup on tape.

To prepare a core image tape, on Level 2 or DISK BASIC. you will nerd the Southern Software utility TSAVE, or TRS TBUG. TSAVE is recommended, because it allows you to work in decimal, not HEX, and to check the saved tape. In this example the memory range you need to save is:

43524 to 49151

Using TSAVE, respond as follows:

```
FILENAME? BACKUP (enter)        (or your own file name)
RANGE? 43520,49151 (enter)
RANGE? (enter)
START? 1740 (enter)     (dummy start address)
Tape records...
Reposition, and type C, to check tape.
```

This tape can be reloaded by:

```
SYSTEM (enter)
*? BACKUP (enter)       (or your own filename)
Tape loads...
*? / (enter)
READY
```

B) Backup on Disk.

Go into TRSDOS (or NEWDOS) from Level 2 by hitting RESET. (This will not destroy the stored image of the compiler).

Type: DUMP ACCEL2/CIM (START=43520,END=49151)

This file can be reloaded under TRSDOS by typing LOAD ACCEL2/CIM
(When you enter Disk BASIC after loading ACCEL2/CIM in the future, be sure to set MEMORY SIZE to 43520, or to your value of M).

C) Backup on Wafer.(EXATRON Stringy Floppy commands assumed).

The address to save is 43520, with length 5632. Because 43520 is not representable as an INTEGER, you will have to type it in modulo 65536, i.e. as -22016.

So type @SAVEn,-22016,5632

This can be reloaded (under Level 2 or DISK BASIC) by @LOADn

ACCEL2 SUPPLIED ON DISKETTE.

There are 4 versions of ACCEL2 on the disk, located in memory at the following addresses:

1) ACCEL2A at 59904 for a 48K system.
2) ACCEL2B at 43520 for a 32K system.
3) ACCEL2C at 47616 to develop programs on a 48K system, for sale on 32K.
4) ACCEL2D at 31232 to develop programs on a 32K or 48K system for sale on 16K.

For the following example use either ACCEL2A or ACCEL2B, according to your memory size. Load the compiler under TRSDOS (or NEWDOS) by LOAD ACCEL2A (or LOAD ACCEL2B). When you enter disk BASIC set MEMORY SIZE to 59904 or 43520.

You will only need to use ACCEL2C or ACCEL2D if you are compiling a program for sale to run on a smaller machine, See the later section on "SELLING COMPILED PROGRAMS".

CREATING A SAMPLE PROGRAM.

Despite the triviality of the following example, it does illustrate most of the mechanics of compiling and saving a compiled program, and it should be followed through to completion.

Under Level 2 or DISK BASIC type

```
AUTO
10 DEFINT I-J
20 FOR I=1 TO 1000:NEXT
30 A$=A$+"*"
40 PRINT J; A$;
50 J=J+1
60 IF J<5 THEN 20
70 STOP
```

List the program, check it, run it, and change it, if necessary. Once you have compiled it, you will no longer be able to EDIT it. So SAVE it on tape, or disk.

COMPILING THE PROGRAM.

To invoke ACCEL2 you must first "activate" the compiler, and then use its builtin commands. Activate it by branching to its first location:

```
SYSTEM (enter)
*? /43520 (or your value of M)
READY
```

The ACCEL2 builtin commands are all BASIC keywords, preceded by a slash (/), (Under NEWDOS80, precede the slash by a blank). To compile, type:

```
/FIX (enter)   (i.e. FIX program in machine-code)
ACCEL2 prints out 5 values, the changing program size.
READY
```

Use of the word FIX is intended to remind you that your BASIC program has now been irreversibly converted to machine-code, You cant EDIT it in any way, but you can LIST it. Shown in comparison with the original, it will look like this:

```
Before Compilation          Compiled by ACCEL2
10 DEFINT I-J               10 DEFINT I-J
20 FOR I=1 TO 1000:NEXT     20 REM
30 A$ =A$+"*"               30 REM
40 PRINT J; A$;             40 PRINT J; A$;
50 J=J+1                    50 REM
60 IF J<5 THEN 20           60 REM
70 STOP                     70 STOP
```

Notes:

1) Machine-code lines appear in the compiled program as REMarks. (The actual machine-code itself follows the REMark, but is unprintable).

2) I and J were defined as INTEGERS in line 10, and as a result the machine-code compiled by ACCEL2 will be much faster than if they had been float variables (SINGLE or DOUBLE).

3) ACCEL2 compiles line 30, the STRING assignment, although ACCEL would not.

4) DEFINT, PRINT and STOP were not compiled, but the run-time environment is smart enough to ensure that the BASIC interpreter is passed control for these statements, and that its understanding of the variables, A$ and J is the same as that of the compiled code.

RUNNING THE COMPILED PROGRAM.

```
RUN (enter)
0 * 1 ** 2 *** 3 **** 4 ***** (program runs)
BREAK IN 70
READY
```

A second RUN will rerun the program, GOTO 10 or GOTO 20, etc. will reenter the program without resetting J to 0 or A$ to null. Notice that the compiled program runs much faster than the original, but is compatible in other respects. RUN it again, but hit BREAK to interrupt the program before completion. Note that the interrupt "takes" in line 40, not one of the machine-code lines. Type ?I;J;A$ to interrogate the current values of the variables. Type A$="A", and then CONT to continue execution, with a modified value of A$. Turn trace on by typing TRON, and rerun the program, Only the uncompiled lines are traced. Turn trace off again with TROFF.

Once you have compiled a program, you can no longer use the commands EDIT, AUTO, CLOAD?, CSAVE, DELETE, MERGE or SAVE. This is because the Machine-code in the compiled lines may contain bytes that are treated as control codes by the interpreter. So use of these commands may cause an infinite loop, or a machine reboot. To get the machine back to its normal, editable state, you must use NEW or CLOAD, or in DISK BASIC, LOAD or RUN "program-name". All of these destroy the compiled program.

SAVING THE COMPILED PROGRAM.

A) On tape:

Saving the compiled program on tape has the merit that the single tape file will be self-contained, Using the SYSTEM command, you can load the tape at a later date, or on another TRS-80, and the program will run, independent of the compiler. However, to save the tape you will need Southern Software's TSAVE. (TBUG is not satisfactory). Relocate TSAVE in a separate area of protected memory, prior to compilation, or build yourself an absolute-address copy of TSAVE to load on top of the compiler (not the run-time routines) after compilation,

Give the following responses:

```
FILENAME? SAMPLE        (or your own file name)
RANGE? 16512,16863      (save control storage)
RANGE? 16548↑,16635↑    (save the compiled program)
RANGE? 43520,44799      (save the run-time routines)
RANGE? (enter)
START 1740              (dummy start address)
```

Notes:

1) Locations 16512 to 16863 contain control information such as program start and end address, dictionary size, etc. MEMORY SIZE is also implicitly set, when this tape is reloaded, to what it was when saved,

2) 16548↑ to 16635↑ means save the range defined by the values contained in these locations. This includes the compiled program and its dictionary of current variables, but excluding the array variables.

3) To run the compiled program you must have available the ACCEL2 run-time routines, (the routines that interface with BASIC), and these routines must be at the same location in memory as they were when the program was compiled. For ACCEL2 these routines constitute the first 1280 bytes. So the values in this third range depend on where you have loaded the compiler, You need not save this range if you know the compiler will already be loaded, when you want to rerun the compiled program.

4) Tape is a cheap and fairly satisfactory way to distribute compiled programs for sale, and the control values saved in the first range will ensure that the program will automatically protect its own run-time component, even if your customer forgets. If, as is likely, you are compiling programs on a large memory, for sale to users with a smaller memory, e.g. 16K, then you need ensure only that the bottom 1280 bytes of ACCEL2 lie within the 16K, i.e. the bulk of the compiler can reside above this boundary.

5) On Video-Genie, use the ESCAPE key for upward arrow.

B) On Disk:

ACCEL2 contains its own built in routines to save the compiled program on disk.

```
/SAVE "PROGRAM" (enter)   (or any valid FILESPEC) will save the compiled program.
/LOAD "PROGRAM" (enter)   will reload the saved program.
/RUN "PROGRAM" (enter)    will load and run it.
```

There are complex rules concerning the name of the saved program, and how these can be used from another compiled or uncompiled program. These are described later.

RELOADING the COMPILED PROGRAM.

A)From Tape:

Type:

```
SYSTEM (enter)
SAMPLE (enter)   (or your own file name)
Tape loads...
*? / (enter)
READY
RUN (enter)
0 * 1 ** 2 *** 3 **** 4 *****    (program runs)
BREAK IN 70
READY
```

B)From Disk:

You must load the compiler first under TRSDOS (or NEWDOS). If you dont want to do any compilations, then you can produce a core-image file which is only the first 1280 bytes of ACCEL2, and load this instead. But you must also allow a further 256 bytes for the disk loading and saving buffer immediately above this. I.e the full run-time component when using disk occupies the first 1536 bytes of the product.

When loading a compiled program from disk, you must ensure that the environment is the same as it was when the program was compiled and saved, namely:

- MEMORY SIZE the same.
- NUMBER OF FILES the same.

In DISK BASIC, with the run-time component loaded, you can use ACCEL2's built in commands to load or run a program. To do this, ACCEL2 must be activated, by branching to its starting location. However do not activate it if it is already active. (This will cause a REBOOT).

```
/LOAD "PROGRAM"   will load the saved program into memory.
/RUN "PROGRAM"    will load and run the program.
```

When loading or saving compiled programs you may encounter various disk errors. These are either reported in the usual way by DISK BASIC, e.g. FILE NOT FOUND, or if they are detected by ACCEL2. as

```
FILE ERROR CODE: n
```

The value of n is as documented under TRSDOS Error Codes in section 6-12 of the TRSDOS & DISK BASIC reference manual.

MORE ON COMPILER ACTIVATION.

The TRS-80 Level2 code in ROM provides (very intelligently, and with a great deal of fore-thought) a table of transfer addresses (in RAM) through which flow passes at certain key points of execution. ACCEL2 uses 3 of these to get control in the following situations:

1) At the beginning of execution of each program statement.
2) At the beginning of execution of each direct command.
3) During the execution of RUN, NEW, CLEAR and END.

When you branch to the starting address of ACCEL2 it enables the first two of these. (The third is handled internally). Because ACCEL2 then gets control on each command or statement, it is able to support new commands of its own, which it chooses to distinguish with a / prefix. Many other products use a similar technique. ACCEL2 attempts to coexist with these products by preserving the original transfer address, during activation, and branching to it, when it has finished its own work. Other products you may want to use in conjunction with ACCEL2 may not be so kind, but may simply overwrite the original transfer address with their own, thus "disabling" any other product playing the same trick. If you encounter this problem, solve it by activating ACCEL2 last.

Inadvertent reactivation is ignored by the compiler. However, once a switch has been enabled, it would be a disaster if the compiler is destroyed in memory while the switch is still active. So ACCEL2 supports a command /RESTORE which will reset the transfer values to their original values, i.e. will deactivate itself. Use this before you overwrite ACCEL2 with another program. Otherwise you can leave it active indefinitely.

COMPILING EXISTING PROGRAMS.

Having tried out the mechanics of compilation on the sample program, you will now want to apply ACCEL2# in earnest, to a real application. Before doing so, however, you should refer to the restrictions at the back of this document. You may have instant success with your own programs. This depends on how tidy and structured a programmer you are. The compiler relies on your programs obeying certain BASIC language rules which are not enforced by the BASIC interpreter, of which the most important are:

- The variables must be named in a way that is consistent, whether the program is compiled from top to bottom, or whether the program is interpreted dynamically.
- The FOR-NEXT loops must be properly nested and structured, both statically and dynamically.

In one independent sample of 72 BASIC programs only 2 failed, due to bad FOR-NEXT structuring, and none due to inconsistent naming. However this sample was published, educational material, written in a plain, explicit style, If your style is less disciplined, then you may suffer a greater percentage of first-time failures.

1) Check your program for CLEAR statements. These reset the meaning of any DEF or DIM statements, and can lead to inconsistent names. If possible, ensure the program starts with CLEAR, and has no other CLEAR statements. Preferably, use explicit DIM statements for all arrays.

2) Check that each FOR statement pairs with one and only one NEXT statement, Watch for RETURN statements which exit from an uncompiled FOR loop, or GOTOs out of FOR loops which could have the same effect.

CHAINING PROGRAMS.

ACCEL2 allows you to chain programs together, i.e. to proceed through a sequence of routines, each invoking the next from disk, and being overlaid by it. The chained programs may be either compiled or interpreted, or a mixture. You will need to debug these segments in an arbitrary order, compiling each one after it is checked out, and you will not want to change the chaining program, when the program it chains to is compiled.

The way this is achieved is as follows: /SAVE filespec saves the current compiled program. If filespec contains no file extension (/ qualifier), then the program is saved with this name. I.e. SAVE "PROG" will overwrite any file called PROG, whether it is a source program or a compiled program. But if the file has an extension then
a) It must be a 3-character extension, and
b) /SAVE will implictly change the extension to ACC.

If your source program is called "PROG/BAS", for example, then after compilation, /SAVE "PROG/BAS" will save the compiled program as "PROG/ACC". (It will also save e.g. "PROG/NEW" as "PROG/ACC" as well, so be careful only to use one extension type for programs you intend to compile).

/RUN filespec or /LOAD filespec also work in a similar way, namely, if the filespec has no extension, then it is treated as the name of a compiled program, and that file is loaded. (If the file is not a compiled program, then an error will result), If, however, the filespec contains an extension then
a) It must be a 3-character extension, and
b) ACCEL2 will attempt to load the file with the extension ACC, but
c) If no such file exists, then ACCEL2 will pass control to the BASIC interpreter causing it to LOAD or RUN the program as a BASIC source program.

These rules apply whether the filespec is a constant (as shown) or whether it is an expression, and they apply if the /SAVE, /LOAD or /RUN statements are executed from the keyboard as direct commands, or if they are within a program (which in turn can be compiled or uncompiled). The filespecs can also include password or drive-number qualifiers.

These rules enable you to debug and incrementally compile a suite of programs provided
a) You use 3-character extensions on your program filenames.
b) You replace all RUN filespec statements in your programs by /RUN statements.

If this aspect of ACCEL2 is important to you, follow the next example through:

```
PROG1/BAS:
10 FOR I%=1 TO 5000:NEXT
15 PRINT"PROG1"
20 /RUN "PROG2/BAS"

PROG2/BAS:
10 FOR I%=1 TO 5000:NEXT
15 PRINT"PROG2"
20 /RUN "PROG1/BAS"
```

Save these two programs on disk (without compilation) as "PROG1/BAS" and "PROG2/BAS". RUN either, and they will loop for ever, each calling the other. Of course you must have ACCEL2 active while they are running, or the BASIC interpreter will reject the /RUN statements, Interrupt the sequence by hitting BREAK.

Now compile one of the programs, PROG2 say, and save it by /SAVE "PROG2/BAS", Again RUN either program, and they will loop as before, PROG1 running interpretively, and PROG2 compiled. (You can tell this from the relative time they take to execute). Finally compile the other program, PROG1, and /SAVE "PROG1/BAS". Now either RUN or /RUN either program to recreate the loop, this time with both programs running compiled.

Notes:

1) The filespec name on disk is of course "PROG1 /ACC" for the compiled program, so RENAME, COPY or KILL, etc., must use this name,

2) If you save a file "X" under TRSDOS, it will normally find "X", if it exists, and write the new file on top of the old, on the same disk drive. If it cant find "X", it writes the file on drive 0. These rules apply to the name "X/ACC" in /SAVE, not to "X/BAS".

SELLING COMPILED PROGRAMS.

One of the major attractions of a BASIC compiler is that it enables you to write BASIC programs for sale which, with care and tuning, can be comparable in performance with machine-code programs, Secondly, and no less important, a compiled program is very difficult to steal. It can be copied, of course, since any file can be copied byte for byte, but it cannot be modified, except by the owner of the original source BASIC. And of course you dont have to release this when you sell a compiled program.

Although tape is an unpopular medium, it has a number of very significant advantages. Cassettes are very cheap, and therefore expendable or replaceable. They are small and light to post, and will survive violent handling, unlike diskettes which need a lot of protection. Finally the TRS-80 built in SYSTEM command is part of ROM, and therefore consistent on all machines, and it is powerful enough to load any number of core-image segments directly into RAM, without restriction.

So if you can ship on tape, do so. The procedure for creating sale tapes is exactly the same as that described in saving the compiled example program on tape. Use the addresses in the earlier table to minimise your use of the customers memory to only the run-time component of ACCEL2. You have to save the three ranges

a) Control storage, (including program size, memory size, etc).
b) The program itself, including its dictionary of scalar (nonarray) variables.
c) The ACCEL2 run-time routines which interface the running program to interpretive BASIC.

Whether on tape or disk, do NOT save the whole of ACCEL2. or you will be regarded as infringing the copyright. Also, you must give an acknowledgement in your program documentation that it was compiled by Southern Software's ACCEL or ACCEL2.

On disk the situation is not so simple. The DUMP routine provided under TRSDOS (or NEWDOS) will only save a single contiguous core image, and it refuses to save any range below HEX'7000'. These two restrictions make it more difficult to sell a compiled program as a single file on disk. What you must do instead is to /SAVE the compiled program as described, as a single file, "PROG/ACC" say, and also to DUMP on the sale diskette the core-image of the run-time component of ACCEL2, as a separate file, LOADER/CIM, say. (Again, do not save the whole compiler). This core image is the first 1280 bytes of wherever you have located ACCEL2, but you must also allow a further 256 bytes of I/O buffer above this in the purchaser's memory.

As an example suppose you want to sell a program "PROG/ACC" to run on a 16K machine (although you have a 32K machine). The full sequence is as follows:

1) The required location for ACCEL2 is 32768-1280-256 = 31232. Under Level2 set this as MEMORY SIZE, load the original selfrelocating version of ACCEL2, and locate it at 31232. (If you have purchased ACCEL2 on disk, then there is already a version at this location).

2) Return to TRSDOS and DUMP this version of ACCEL2 as a core-image file for your own use.

3) Enter DISK BASIC setting the NUMBER OF FILES to whatever will be required by PROG/ACC, and the MEMORY SIZE to 31232 again.

4) LOAD the source for PROG/ACC, compile it, and then /SAVE the compiled program as PROG/ACC as described earlier, but onto a new master disk.

5) Return to TRSDOS and DUMP the run-time component of ACCEL2 on this new master disk, as a file called LOADER/CIM, say. I.e. DUMP LOADER/CIM (START=31232,END=32511).

6) This master disk will now contain two files PROG/ACC and LOADER/CIM, not the full core-image of ACCEL2. TRS BACKUP is now a convenient way of making copies of this disk for sale.

Your operating instructions must now include the following directions to the end-user. (Alternatively, you can automate the procedure with the use of Southern Software's Command-List processor, EXEC).

1) From TRSDOS load the run-time routines by LOAD LOADER/CIM.

2) Enter DISK BASIC setting NUMBER OF FILES to N (the number you used earlier) and MEMORY SIZE to 31232.

3) Activate the loader by SYSTEM (enter) and *? /31232.

4) Load the compiled program by /LOAD "PROG/ACC", and follow this by RUN. Or use /RUN "PROG/ACC". (It is unnecessary to confuse the issue by saying that /RUN "PROG/BAS" would also work).

EXECUTION PERFORMANCE.

The aim of using a compiler is to improve execution speed. But the compiler cannot do better than the machine on which the program runs. The Z80 CPU chip is remarkably cheap, reliable, and fast, but it lacks many common operations (such as multiply and divide), These have to be executed via calls to ROM routines which provide the required function (e.g. multiply by successive additions), and this is of course relatively slow. The complex table at the end of this section is a guide to what features can be improved by compilation, and by how much. It remains one of the programmer's tasks (unfortunately), to match the requirements of the problem to the capabilities of the underlying computing system, The extra effort needed to optimise performance could be thought of as a form of machine-code programming. It can produce results comparable in performance with real assembler language coding, but it is incomparably easier, because debugging is in BASIC, using PRINT statements, TRACE, etc.

The result of compilation is a program which is a mixture of BASIC statements and directly executing Z80 machine-code instructions. The Z80 can execute branches and subroutine calls, and can perform logic and arithmetic (excluding multiply and divide) on INTEGERS, but not on SINGLE or DOUBLE precision floating-point numbers. Nor can it directly manipulate the internal form of BASIC strings, although it can move strips of bytes from one variable to another quite efficiently. (The difficulty with strings is that their lengths vary dynamically). With the exception of SET and RESET, ACCEL confines its translation to those operations that can be expressed directly in machine-code, but ACCEL2 also translates many statements to sequences of calls to routines in ROM, or to its own run-time component.

In addition to the actual execution of the program operations, there is the "resolution" of the variable names and line-numbers, Here A compiler comes into its own. The BASIC interpreter resolves each name by a sequential search through its dictionary (table of variables), every time the variable is referenced during execution. In contrast the compiler allocates storage for the variable once during compilation, and then replaces each compiled reference by a direct machine address, rather than a dictionary search. Similarly each reference to a line number in GOTO or GOSUB translates to a simple branch address, whereas the BASIC interpreter has to search the program sequentially from the top to find the target line.

One effect of BASIC's two forms of sequential search is that the running time of a program depends on how large it is. The more variables you have in your program. then the longer the average time taken to find each one, and the more lines in your program, the longer it takes to execute each GOTO or GOSUB. The speed of the compiled code, on the other hand, is independent of program size and number of variables. This means that it is quite impossible ever to make a firm statement about relative performance, since you cannot say how long a statement such as A = B + C will take under the interpreter. It depends on context. Similar arguments apply to program size before and after compilation. Programs may contain REMarks and blanks. BASIC names can be any length, After compilation all these uncertainties disappear - the REMarks and blanks are removed (from translated code) and the variable and line references are all two-byte addresses.

So the table that follows is in one sense very pessimistic. The timings were all taken on the smallest program in which they could be measured, i.e. a simple FOR-loop. There were no blanks or remarks in the source, and the names were all two bytes long. The performance improvement measured for GOTO, for example, is 176 to 1, but in a large program this could be even greater. But the catch is that this figure may be irrelevant. Because the directly executing operations are so fast, they scarcely contribute to the execution total at all, and performance becomes dominated by those operations that are not compiled, e.g. READ, by the out-of-line subroutines, e.g. Multiply, or by I/O.

Apart from indicating performance gains the table also summarises those language features that ACCEL2 will translate, rather then leave to BASIC. CLEAR, RUN and RESTORE are also translated, but for expediency rather than as a performance improvement. Anything else not shown in the table will cause the whole statement in which it appears to be left in its interpretive form.

SPEED/SPACE Performance Table.

Speed Improvement(Ratio)				operation	Space Degradation(Bytes)			
INT	SNG	DBL	STR		INT	SNG	DBL	STR
121.0	3.7	3.2	7.9	Assignment (LET)	5	14	14	14
29.5	83.8	71.5	30.4	Array Ref (1-dim.)	16	24	25	20
48.2	3.1	2.5		AND and OR	5	14	14	
55.3	2.8	2.3	14.9	Compare (=)	11	26	25	10
61.8	2.8	2.0	2.5	Add,Concat (+)	0	2	2	1
50.6	2.7	2.0		Subtract (-)	3	2	2	
2.5	3.2	2.0		Multiply (*)	5	2	2	
1.1	1.2	1.02		Divide (/)	5	2	2	
83.5	22.9	85.7	2.1	Constant Reference	0	6	10	7
11.4				FOR-NEXT	29			
80.6	5.7	4.2		POKE	7	19	19	
7.7	3.7	3.1		SET and RESET	6	18	18	
17.7	2.7	2.2	9.9	IF-THEN-ELSE	15	21	21	21
17.4	3.7	3.1		ON expr GOTO	12	18	18	
				Function				
5.6	6.4	6.6		POINT	3	9	9	
inf	inf	inf	inf	VARPTR	-3	-9	-9	-9
128.0	inf	inf		PEEK	0	0	0	
			56.5	LEN				1
			4.7	MID$				5
			3.2	LEFT$				4
			3.1	RIGHT$				4
			3.4	CHR$				2
			25.4	ASC				7
			24.9	CVI				8
148.8				GOSUB-RETURN	4			
176.4				GOTO	0			

Disclaimers:

1) No commitment is implied by these figures. They are subject to all sorts of variability (e.g. time to reference a constant depends on the value of the constant being referenced).

2) Speed ratios for STRINGS depend on length of strings, whether the string is a program constant (a literal), whether the receiving string is tire sane length as the source string, etc. Four-byte strings were used in measurements.

3) Use of "inf" (infinity) in the table means that the ratio could not be measured meaningfully. E.g. the reference VARPTR(X) in interpreted BASIC always takes longer than the reference to X. But in compiled code the reference VARPTR(X) actually takes less time than the reference to X, if X is anything other than INTEGER.

4) Negative numbers in the space table mean that the compiled code occupied less space than the original.

PERFORMANCE HINTS.

Nothing the compiler can do will speed up I/O devices - disk, tape, printer, or keyboard. But for processing limited by CPU speed, the following are good rules:

1) Always use INTEGER data types whenever possible, since these are the only data elements the CPU can manipulate directly. You can qualify variable names with % to make them INTEGERS, but better is to get into the habit of coding e.g. DEFINT I-P at the head of each program. In particular use INTEGERS for any FOR-loop control variable.

2) Avoid continually processing DATA with READ statements. Rather, READ the data values once into an array and process from that. This avoids the very considerable overhead of converting the DATA constants from character to numeric on every use.

3) Translation of a statement from BASIC to machine-code is often prevented by the existence of a single non-compilable operation or function. Check the compiled listing to see which statements have been turned into REMarks. Then isolate non-compilable functions (or move them out of a loop) in order to minimise their effects. Sometimes alternative techniques can be used. E.g RND is a very expensive function indeed, and its use can often be circumvented.

4) Compilation of string expressions may be limited by complexity. In this case break the statement down.

5) There is a well-known execution "hiccup" caused by string space "garbage collection", (recovery of free space). ACCEL2 does not affect the actual garbage collection process, but it does attempt to minimise its frequency of occurrence, by avoiding string space allocation if possible. In particular, if string sizes match in assignment, then a spectacular improvement may result.

COMMON PITFALLS.

1) Many programs have loops that are simply there to delay the process, e.g. to make a "ball" moving on the screen go more slowly. Either lengthen these loops when the program is compiled, or alternatively use SINGLE or DOUBLE control variables for the loop, since these types of FOR-loops are not translated by the compiler.

2) 100 GOTO 100 is a common way of terminating a program to avoid the READY message corrupting the screen. This loop cannot be interrupted by the BREAK key, and will need RESET. Instead use

```
100 RANDOM:GOTO 100
```

3) If you choose different MEMORY SIZE settings from the example given in the text, or if you position the compiler elsewhere in memory, then be sure the address arithmetic is correct. This is very error-prone. Work it out on paper first, and type it in from the written copy.

4) When you have compiled a program, do not use the editing commands, since they will produce completely unpredictable results. Always reset the machine state with NEW, LOAD, or CLOAD.

5) It is common practice to use DATA statements as a source of variable data. I.e. after running the program once you EDIT new values into the DATA statements for the next run. This isn't possible once the program is compiled, and an alternative will have to be used, i.e. keyboard input, or files.

6) There is a hardware bug you may suffer from. Run the following program in BASIC:

```
DIM A%(10): PRINT VARPTR(A%(1)) - VARPTR(A%(0))
```

The answer should be 2 (since INTEGER variables occupy 2 bytes per element). If the answer is 3, you have a Z80 CPU bug, and you should compile all programs preceded by the REM NOARRAY option (see below).

COMPILER OPTIONS.

ACCEL2 supports 2 compile-time options which control the level of translation,

EXPR or NOEXPR	Optional compilation of expressions.
NOARRAY	Suppress compilation of arrays.

The EXPR option can be set on or off at any point in the program by inserting a REMARK with the control word after it, i.e.

```
REM NOEXPR   or
REM EXPR
```

Each must be on a separate line, and each must be terminated by ENTER (Video Genie NEWLINE), with no trailing spaces.

REM NOEXPR will inhibit compilation of expressions, and hence of LET (assignment), FOR, POKE, SET, RESET, and IF expressions. (GOTO, GOSUB, RETURN, ON and THEN/ELSE decisions are always compiled). When compiling a large program for the first time there is no harm in setting REM NOEXPR for the whole program. That way it is less likely to exceed your memory. If this succeeds, build up the degree of compilation gradually.

The NOEXPR option enables you to reduce the compiled program size, at the expense of less performance improvement. Typically a small part of any program will be responsible for the majority of the execution delay, whereas initialisation sections, and exceptional conditions or error handling are not at all critical. Bracket the performance-critical sections with:

```
REM EXPR
performance-critical section
REM NOEXPR
```

In this way you can maximise performance, while minimising code expansion.

REM NOARRAY is used in the same way as REM NOEXPR, except that it must apply to the whole program, i.e. use it at the front or not at all.

The TRS-80 supports adjustable-bound arrays, e.g.

```
   10 INPUT N:DIM A(N)
or
   10 DIM A(10):...
   20 CLEAR:DIM A(20)...
```

ACCEL2 cannot cope with this degree of flexibility, and is unable to compile programs containing references to such arrays, i.e. the program will fail. If you have such arrays you must precede the program with REM NOARRAY.

COMPILE-TIME MESSAGES.

These are messages you may get when compiling a program with ACCEL2,

OM OUT OF MEMORY. Compiler could not complete.
FC ILLEGAL FUNCTION. Disallowed statement, e.g. CSAVE.
NF NEXT WITHOUT FOR. FOR-NEXTs not statically matched.
LS STRING TOO LONG. Machine-code expansion exceeds 256 bytes. Simplify the line.
UL UNDEFINED LINE. Bad line number referenced in GOTO or GOSUB.
CN CANT CONTINUE. Compiler cant parse the statement. Check the line for a SYNTAX error.

Note that ACCEL2 passes "illegal" statements through unchanged, e.g. @SAVE 1. This permits the use of stringy floppy control statements in a compiled program (or any of the many extended statement languages available today).

During compilation 5 numbers are displayed. These are put out chiefly as an aid to see how compilation is progressing. The first is the size (in bytes) of the original BASIC program. The next 4 are the sizes of the program after each of the four compiler passes.

PASS 1 builds the variable dictionary, and modifies some of the source statements, e.g. DATA statements are moved to the back. It removes REMarks, so the program size will usually go down.

PASS 2 maps out exactly what code will be compiled so that the next pass will know where each line will eventually be in memory. It does not change the program.

PASS 3 actually compiles the code, and is the slowest pass, and the one that expands the text.

PASS 4 tidies up, removing flags used internally by the compiler from the program.

Any diagnosed error will stop the compiler, and leave the program half-compiled. Dont attempt to correct the error by editing. Type NEW and reload the program. (If NEW results in SYNTAX ERROR, then compile this null program, and that will clear the problem).

RESTRICTIONS.

Experience of early users of ACCEL and ACCEL2 has shown that some programs working under BASIC may fail in execution, or even in compilation. These failures were almost always due to the program infringing one or more of the restrictions below, rather than as a result of a compiler bug. So if you encounter a problem, believe that it is as a result of a restriction, and identify the problem by tracing the program, inserting diagnostic PRINT statements, or by breaking the program down into segments.

Certainly users who create a program from scratch, compiling occasionally as they go to check progress rarely suffer any real constraints. ACCEL2 relies on your program obeying certain BASIC rules which are not checked by the interpreter, and indeed are not necessarily documented in the TANDY manual. An example is the "strength reduction" applied by the interpreter. If two INTEGER variables are added, their result may exceed the maximum INTEGER size (32767). In this case the interpreter turns the temporary result into a float (SINGLE) variable. No compiler can afford to produce code to check for this contingency, and indeed it is certain that the original BASIC language designers would have regarded this as an error. Unfortunately the effect here is that compiled and uncompiled programs can give differing results.

1) No redefinition of meaning of names.

The names in your program must mean the same whether the program is read globally as the compiler sees it, or executed dynamically, as the interpreter sees it. E.g.

I=1 .... DEFINT I:I=1 is disallowed. (The interpreter will treat the first I as SINGLE).

You are unlikely ever to do this sort of thing deliberately, but it can come about, e.g. if CLEAR is used other than at the top of the program. CLEAR resets variables types to default (SINGLE), and may therefore cause a variable to change from INTEGER to SINGLE without your meaning it to.

2) Dimensions must be constant.

ACCEL2 cannot compile programs which contain either

```
DIM A(N)
```

or

```
DIM A(10) ...
CLEAR
DIM A(20) ...
```

In either of these cases the program must be preceded by REM NOARRAY. Of course this inhibits optimisation, and if possible it would be better to change the original program to use fixed bound arrays.

3) The array dictionary is built dynamically.

This is another, more subtle flavour to the above restrictions. When a compiled program is saved on tape or disk the scalar (non-array) part of the dictionary is saved with it, So when it's reloaded that dictionary is already in existence before you say RUN. But arrays are sometimes very large, and saving them in this way would make the saved files unreasonably big. So instead ACCEL2 relies on the run-time execution recreating the arrays again in exactly the same place as they were when the program was saved. I.e. ensure that the flow into your program passes through the DIM statements before any array element is used, or ensure that all arrays that do not have DIM declarations have their elements first referenced in the same order as the compiler will see them (top-to-bottom). Do not code e.g.

```
10 GOTO 30
20 A(1)=1
30 B(1)=2
90 A(1)=3
```

since the compiler will see A before B, whereas execution will see B before A. Ideally, supply DIM statements at the front of the program (but after CLEAR) for all arrays.

4) FOR-NEXTs must be properly structured, statically and dynamically.

Each FOR statement must belong with a unique matching NEXT. Two FORs cannot share the same NEXT, and two NEXTs cannot share the same FOR. More difficult to spot are cases of bad run-time nesting. The compiled code and the BASIC interpreter share the same run-time stack, and this stack, is used by compiled code for RETURN, FOR and NEXT, The BASIC interpreter uses it for any uncompiled FOR-NEXTs. If you exit from a FOR-loop without closing it (e.g. you branch out of it, or RETURN out of it) then the stack entry for the FOR-loop is not cleared. ACCEL2 is smart enough to pick up RETURN out of an unclosed compiled loop and to close the loop automatically. It cannot handle the case of an unclosed, uncompiled loop however and this may cause a wild branch, or a reboot. Also, it cannot trap GOTO out of a loop, because the flow may branch back in, and continue from where the loop left off. To be safe, code as follows:

```
OK                              BAD
10 FOR I=1 TO 10                10 FOR I=1 TO 10
20 IF I=5 THEN I=10:GOTO 50     20 IF I=5 THEN GOTO 60
30 ... (body)                   30 ... (body)
50 NEXT                         50 NEXT
60 OK, closed                   60 NOT OK, open
```

If you compile a working program and it fails wildly, then bad nesting is by far the most likely cause. Running with trace (TRON) will usually identify the problem.

5) Error behaviour is not necessarily consistent.

INTEGER overflow is nut diagnosed. E.g. IF A>B THEN may fail if A-B is greater than 32767, or less than -32768. Out-of-range arguments to string functions (e.g. MID$ offset and length) are rounded modulo 256. Values out-of-range in ON statements are treated as zero, not errors. Out-of-memory may nut be diagnosed at run- time, and may cause a wild branch, or a reboot. Your program may contain errors which BASIC does not diagnose, but which the compiler will reject, for instance bad syntax in an ELSE clause which is never executed.

In general, programmed error handling (i.e. the use of ON ERROR) is unlikely to work. This is firstly because the error you are trying to trap may not be caught by the compiled code at all. But also the target line N of ON ERROR GOTO must be an uncompiled statement, or the program will loop on the error. If you have ON ERROR GOTO N anywhere in your program, then put a colon (i.e. a null statement, ":") at the front of line N.

6) Current line-number not maintained.

Lines which start with statements that have been compiled to machine-code do not update the current line number, which therefore remains set at the last executed non-compiled line. Therefore BASIC diagnostic messages may be misleading, and RESUME (without line number) may possibly fail by branching to the wrong line. TRON will give an incomplete trace. The BREAK key will only interrupt in noncompiled lines.

7) Compiled programs may not be EDITED.

When the machine holds a compiled program you may not use the commands EDIT, AUTO, CLOAD?, CSAVE, DELETE, MERGE, and SAVE, and obviously these must not appear in a program you try to compile, (This gives an ILLEGAL FUNCTION diagnostic), In addition GOSUB and REMARK should not be used as keyboard (i.e. direct) commands.

8) No code expansion during compilation may exceed 256 bytes.

This gives a STRING TOO LONG error, and the offending line should be broken up.

9) The name QX% is reserved. (QX% is used in the compiled code as a temporary variable).

10) Disk and Wafer operations are mutually exclusive.

To save space, ACCEL2 puts the wafer saving and loading routines in the area used as a 256-byte buffer for disk SAVE and LOAD. Once you have used /SAVE, /LOAD, or /RUN, then you cannot use /@SAVE, /@LOAD, or /@RUN.

11) USR calls are only optimised under Level 2 or TRSDOS2.3, not other operating systems.

southern software

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

How to Load and Relocate a Southern Software Machine-Language Program.

You choose the location of the program in memory, to suit your machine size. This MUST be in protected memory, or the program will not run. So, taking account of your machine size, allow enough space for the program itself, plus any other machine-language subroutines you may need, either above or below the program you are loading.

As an example, suppose you are loading Southern Software DLOAD (size 160 bytes). You have already loaded, or are going to load, TRS KBFIX at the top of memory, and Southern Software TSAVE below DLOAD. Plan your memory use as follows, working out the values (T) and (A) for your situation:

	PROG SIZE (bytes)	MACHINE SIZE 4K	16K	32K	48K	
Memory limit		20480	32768	49152	65536	
Space for KBFIX	56	20424	32712	49096	65480	
Space for DLOAD	160	20264	32552	48936	65320	(T)
Space for TSAVE	512	19752	32040	48429	64808	(A)

1)Turn on the computer. If you have a DISK system, enter Level2, not DISK BASIC.
2)answer the MEMORY SIZE question with your value of (A). (On Video Genie, this value is used after READY?).
3)prepare the cassette player to load the self-relocating program.

	TRS-80	You type:
4)	>	SYSTEM (enter)
5)	*?	DLOAD (enter) or your program name
6) After tape has loaded	*?	/ (enter)
7)	TARGET ADDR?	Your value of (T)
8)	READY	

Notes:
1)At step 5 the tape will load and a pair of asterisks will blink on the display. If there are no asterisks, or two unblinking asterisks, or C*, then there has been a loading error. Stop the recorder, reset, and retry with a new volume setting.

2)At step 7, the program will relocate itself to address T. If instead of typing a value you just hit enter, then the program will relate itself to A, the answer to the MEMORY SIZE question.

3)Under Level2, after relocation, the program is ready to be invoked with a USR(n) call, since the USR address is automatically primed. However this does not work under DISK BASIC (or Level3), and you must additionally set DEFUSRn to inform the system of this routine's address.

4)Once a program has been loaded and relocated, it can be dumped to a new tape using Southern Software TSAVE, or TRS TBUG. Then it will load directly to its final location. Use of TSAVE has the advantage that several programs can be dumped on a single file, which can also preprime the USR address.

5)During step 5, the program is temporarily load into locations 18944 and up. This means that
a)You must perform all necessary relocating loads before loading a BASIC program, or entering DISK BASIC.
b)The final location, T, of the self-relocating program can never be lower than 18960. (Hex 4A10).

6)If you run under DISK BASIC, then perform the initial self-relocating load under Level2, as described. Then reenter TRSDOS (or NEWDOS, etc) and use the DUMP command to save the core image directly from its relocated position. Subsequently you can LOAD the core image directly, under TRSDOS. But when you enter DISK BASIC, remember to set MEMORY SIZE to leave this area of core protected, and remember that the top 64 bytes of memory are corrupted by the DISK BASIC loader, and should not be used for programs.

Hints on Tape Loading.

1) Listen to the tape to establish exactly where the data starts. Note this on the tape label.
2) Turn the volume down to zero, and "attempt" a tape load, very slowly increasing the volume until you get asterisks on the screen. Stop the tape (not the computer), note the volume level, Reboot.
3) Turn the volume up to maximum, and "attempt" a tape load, very slowly decreasing the volume until you get asterisks on the screen. Again, stop the tape, and note the vole,
4) Set the volume to slightly above the mid-point of the two extremes of volume, and attempt a real load.

Possible Tape or Recorder Faults.

1) Kink or fold in the tape. Even a minor fold may render the tape unloadable, (Southern Software tapes carry a second copy of the file, in case the first gets damaged).
2) Noise caused by RESET when tape is running, Always stop the tape before hitting RESET.
3) Being small, all the plugs are prone to intermittent error and should be protected against movement.
4) Inconsistent tracking of the tape over the head.

This list does not include poor tape quality, since it is very unlikely to be a problem, at the frequency bits are recorded. However, you may have found that one make of cassette seems much better than another. This is probably due to the construction of the cassette, rather than the tape. Generally more expensive cassettes run more smoothly, and therefore reduce the chance of poor tracking of tape over the head.

How DATA is Recorded and Read.

The computer contains hardware to generate an "above-and-below-zero" pulse, as shown below. This is fired by direct program control. The output routine produces one such clock pulse every 500th of a second (by looping). Data ones and zeroes are recorded as pulses halfway between these clock signals, A zero is the absence of a pulse, a one is the presence of a pulse.

The playback logic is analogous to a keyboard "debounce" routine, To read a single bit, start somewhere near (A). Loop, until the hardware recognises a signal, at (B). This is a clock pulse. Now loop until that signal is bound to have died away, and reset the hardware latch, at (C). Now wait an exact length of time, till (D), and listen for another signal, YES, then it's a one, NO, then it's a zero. In either case reset the latch after the sampling time, at (E), and loop again until the next time (A).

As you can see, the TIMER must not be running during either record or playback, since exact looping times are vital. Nor does the logic take time off to test the keyboard for the BREAK key. However tape speed is not ultra-critical, since there is a resynchronisation wires at (A) on every bit.